

C++ ONLINE

UZOCHUKWU OCHOGU

OPEN CONTENT:

C++ FOR HIGH PERFORMANCE
WEB APPLICATIONS

2026

Why Choose C++ for Web applications?

LOW LATENCY

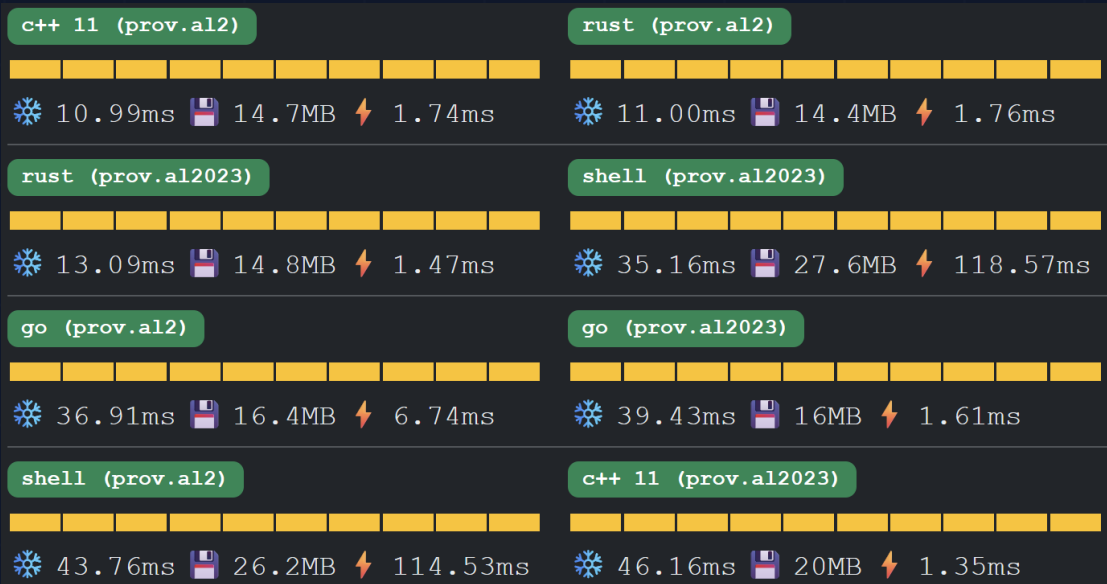
Latency can be taken to millisecond and even sub-millisecond levels where it is bounded by network overhead. Useful for critical APIs and real-time systems.

HARDWARE TUNING & UTILIZATION

Direct access to SIMD, AVX, and hardware-specific optimizations to maximize throughput especially when you have knowledge of execution environments.

CLOUD COST REDUCTION AND BENEFITS

Less compute can serve more. Fast AWS lambda cold start.



```
terminate | kill | signal |
Status: Elapsed: IO/R: IO/W: Parent: User: Threads:
Running 45d 18:09 sh [redacted] 27

Memory: 1.0% ..... 44.3 MiB

./games_bot
```

Why Choose C++ for Web Applications? (Contd)

ACCESS TO POWERFUL LIBRARIES

Implementations of fastest libraries for serialization, json, hash-maps, math, GEMM, SIMD abstraction, searching, string processing libraries etc

e.g `zpp::bits`, `glaze`, `simdjson`, `ankerl::unordered_dense`, `axiom`, `parrot`, `google highway`, `usearch`, `zvec`, `StringZilla`

PERFORMANCE-AWARE PROGRAMMING

You can design data structure, code and access patterns to match hardware assumptions thus minimizing cache misses and maximize memory throughput.

ZERO-COPY NETWORKING

Reducing CPU overhead by-passing data directly between buffers without redundant memory copies.

TECH: RDMA, DPDK, IO_URING/IoRing

DETERMINISM

If determinism is a requirement, stack allocated memory pools, ring buffers can be used to eliminate heap allocations.

CONCERNS

MEMORY SAFETY

MEMORY SAFETY, UB

Risks of manual memory management and undefined behavior are higher than in managed languages. Memory bugs can be exploited.

COMPLEXITY

ENGINEERING COST

Steeper learning curve than regular web applications. Difficult build systems, package management fragmentation. Different “valid” paradigms in C++. Extensive tooling requirements.

MITIGATING SAFETY CONCERNS

- C++ is still not memory safe but common classes of bugs can be eliminated. Understand your application's threat model and design around this.

MEMORY SAFETY

RAII and smart pointers eliminate manual memory management. `std::unique_ptr` and `std::shared_ptr` provide deterministic ownership.

```
auto service = std::make_unique<WebService>();
```

MODERN IDIOMS

Using `std::optional`, `std::expected` for robust error handling, and then exceptions for truly exceptional cases.

```
std::expected<User, Error> find_user(id);
```

STATIC ANALYSIS, SANITIZER, TESTING

Leveraging Clang-Tidy, PVS-Studio, and runtime sanitizers (ASan, UBSan) to catch bugs before deployment. Integration and Unit tests. If necessary, turn on bounds checks.

```
// clang-tidy -checks=modernize-*
```

RESILIENT DESIGN

Rigorous input validation and designing services to be restartable and stateless where possible.

```
validate(input).or_else(handle_error);
```

MITIGATING COMPLEXITY CONCERNS

ADOPT NEWER C++ STANDARD

C++20, C++23, C++26 gives access to `std::expected`, `std::optional`, `std::span`, `std::string_view`, more `constexpr` support.
Concepts simplify metaprogramming. Sometimes ranges algorithms can be more intuitive.

CREATE USEFUL ABSTRACTIONS

Keep utilities headers, organize code into layers. This can aid testing can data flow can be better modelled.

COROUTINES + EVENT LOOP = EASIER CONCURRENCY MODEL

EXTENSIVE DOCUMENTATION

Self documenting code, diagrams, folder level readmes and repository wikis.

HIGH-PERFORMANCE LIBRARIES AND FRAMEWORKS

FRAMEWORK	IO PROGRAMMING MODEL	PERFORMANCE	STATUS
Drogon	stackless coroutines, Reactor, callbacks	Top performer	PRODUCTION READY
Oatpp	Stackless coroutines + Reactor with scheduling	Fast	PRODUCTION READY
userver	stackful coroutines	Top performer	PRODUCTION READY
lithium	stackful coroutines, Reactor	Top performer	STABLE
ffead-cpp	Depends on backend	Top performer	PRODUCTION READY
Boost.Asio & Beast	proactor, callbacks, stackless and stackful coroutines	Top performer	PRODUCTION READY
Metaspex	?	Top performer	PRODUCTION READY

HIGH-PERFORMANCE LIBRARIES AND FRAMEWORKS CONTD

FRAMEWORK	IO PROGRAMMING MODEL	PERFORMANCE	STATUS
Mongoose	Callbacks	Average	PRODUCTION READY
Glaze Networking	futures, Callbacks	?	STABLE
Crow	Callbacks	Average	STABLE
VixCpp	Stackless coroutine, Reactor	Fast	STABLE

DROGON:

REACTOR, CALLBACK AND STACKLESS COROUTINE CONCURRENCY MODEL

Non-blocking I/O for maximum concurrency. Efficiently handles thousands of simultaneous connections per thread. Supports coroutines.

AOP & FILTERS

Support for Aspect-Oriented Programming and powerful middleware and filters for cross-cutting concerns like Auth and Logging.

NATIVE DATABASE DRIVERS

High-performance, asynchronous drivers for PostgreSQL, MySQL, and SQLite integrated directly into the framework. Redis is also supported out of the box.

Extensible, Plugins support

Drogon is extensible. Users can create plugins for SMTP, AMQP (e.g. RabbitMQ), Rate Limiters, metrics and Prometheus integration etc.

DROGON:

```
1 class Auth : public drogon::HttpController<Auth> {
2     public:
3         METHOD_LIST_BEGIN
4         ADD_METHOD_TO(Auth::login, "auth/login", drogon::Post, drogon::Options,
5             "CorsMiddleware");
6         ADD_METHOD_TO(Auth::logout, "auth/logout", drogon::Post, drogon::Options,
7             "CorsMiddleware", "AuthMiddleware");
8         METHOD_LIST_END
9
10        static drogon::Task<> login(
11            drogon::HttpRequestPtr req,
12            std::function<void(const drogon::HttpResponsePtr&)> callback);
13        static drogon::Task<> logout(
14            drogon::HttpRequestPtr req,
15            std::function<void(const drogon::HttpResponsePtr&)> callback);
16 }
```

```
1 #include <drogon/HttpAppFramework.h>
2 int main() {
3     //Set HTTP listener address and port
4     drogon::app().addListener(":", 80);
5     drogon::app().addListener("0.0.0.0", 80);
6
7     // Load config file
8     // drogon::app().loadConfigFile("../config.json");
9     // Run HTTP framework, the method will block
10    // in the internal event loop
11    drogon::app().run();
12    return 0;
13 }
```

```
1 drogon::Task<>
2 Auth::login(drogon::HttpRequestPtr req,
3             std::function<void(const drogon::HttpResponsePtr &)> callback) {
4     auto expected_credentials =
5         utilities::strict_read_json<UserCredentials>(creds, req->getBody());
6
7     // validation logic
8
9     auto db = app().getDbClient();
10    try {
11
12        // DB operations
13
14        UserCredentialsResponse response{
15            .token = token, .refresh_token = refresh_token, .user_id = user_id};
16        auto resp = HttpResponse::newHttpResponse(k200OK, CT_APPLICATION_JSON);
17        resp->setBody(glz::write_json(response).value_or(""));
18        callback(resp);
19        co_return;
20    } catch (const DrogonDbException &e) {
21        SimpleError error{.error = "An error occurred"};
22        auto resp = HttpResponse::newHttpResponse(k500InternalServerError,
23            CT_APPLICATION_JSON);
24        resp->setBody(glz::write_json(error).value_or(""));
25        callback(resp);
26        co_return;
27    }
28 };
```

USERVER

STACKFUL COROUTINES

Handle thousands of concurrent connections without complex async code. Write synchronous-looking code that performs asynchronously.

FEATURE RICH

Async HTTP, gRPC clients, drivers for PostgreSQL (including pipelining and binary protocol), Redis, Mongo, YDB, ClickHouse, MySQL, RabbitMQ and Kafka. Other features like Service Discovery, retries and load balancing out of the box.

USED BY UBER, MATCHMAKER AND YANDEX

Powering massive services like Yandex Go and Yandex Delivery. Proven reliability under extreme production loads.

:: **Observability:** Built-in logging, metrics, and distributed tracing (OpenTelemetry).

:: **Configuration:** Dynamic config management without service restarts.

USERVER

```
1 int main(int argc, char* argv[]) {
2     const auto components_list =
3         components::MinimalServerComponentList()
4             .Append<kafka_sample::ConsumerHandler>()
5             .Append<kafka_sample::ProducerHandler>()
6             .Append<kafka::ConsumerComponent>("kafka-consumer")
7             .Append<kafka::ProducerComponent>("kafka-producer")
8             .Append<components::TestsuiteSupport>()
9             .Append<components::Secdist>()
10            .Append<components::DefaultSecdistProvider>()
11            .AppendComponentList(clients::http::ComponentList())
12            .Append<clients::dns::Component>()
13            .Append<server::handlers::TestsControl>();
14
15     return utils::DaemonMain(argc, argv, components_list);
16 }
```

userver provides a complete toolkit for building modern, scalable microservices with the performance of C++.

```
1 class ProducerHandler final : public server::handlers::HttpHandlerJsonBase {
2 public:
3     static constexpr std::string_view kName{"producer-handler"};
4
5     ProducerHandler(const components::ComponentConfig &config,
6                     const components::ComponentContext &context);
7
8     formats::json::Value HandleRequestJsonThrow(
9         const server::http::HttpRequest &request,
10        const formats::json::Value &request_json,
11        server::request::RequestContext &context) const override;
12
13 private:
14     const kafka::Producer &producer_;
15 };
```

LIBRARY ECOSYSTEM

JSON, XML & SERIALIZATION

Glaze	simdjson	zpp::bits	yyjson
nlohmann/json	pugixml	reflect-cpp	Lite3
RapidJSON	struct_pack	cap'n proto	Cista++
flatbuffers	(yalantinglibs)	msgpack-c	

NETWORKING & IPC

Boost.Asio	gRPC	uWebSockets
ZeroMQ	Thrift	Boost.Beast
nanomsg	POCO networking	

CRYPTOGRAPHY AND SECURITY

libsodium	monocypher
jwt-cpp	libjwt
argon2, bcrypt	

DATABASE DRIVERS

libpqxx	Official C++ client for PostgreSQL.
TinyORM	Modern C++ ORM for SQL databases.
mongo-cxx	Native driver for MongoDB.

EVENT BUS, STREAMING AND MESSAGE QUEUES

cppkafka, modern-cpp-kafka, librdkafka, redpanda

rabbitmq-c, AMQP-CPP, rmqcpp

blazing-mq

IN MEMORY STORAGE

hiredis, redis-plus-plus	- clients for redis, valkey, DragonFly and Pogocache (RESP WIRE PROTOCOL)
Memcached, pogocache	

CLOUD SDKS

AWS, AZURE, GCP

VALIDATION

LEXY, Boost.Parser for input validation

CHOOSING LIBRARIES: WHAT TO CONSIDER

COMMUNITY & ADOPTION

ACTIVE MAINTENANCE

DOCUMENTATION QUALITY

PERFORMANCE METRICS

BENCHMARK FOR YOUR USE CASE

P99 LATENCY

FEATURE ALIGNMENT

EASE OF USE

MODERN C++ IDIOMS

BUILD SYSTEM INTEGRATION

CHOOSING LIBRARIES: WHAT TO CONSIDER

LICENSE

HOW PERMISSIVE IS IT?

Unlicense / CC0 > MIT > BSD > Apache 2.0 > MPL 2.0 > LGPL >
GPLv2 > GPLv3 > Source-Available

→ VERY IMPORTANT FOR COMMERCIAL PRODUCTS

MODULAR, EXTENSIBLE, PLUGGABLE DESIGN

TRANSITIVE DEPENDENCIES

FEATURE COUPLING

SUPPORT EXTENSIONS WITH PLUGINS

DEPENDENCY MANAGEMENT

PACKAGE MANAGER

VCPKG, CONAN, HUNTER

Ensure you lock dependencies to minimize supply chain attack risks!

```
vcpkg install drogon
```

FETCHCONTENT

CMake's native way to download and integrate libraries directly into the build tree.

SOURCE VENDORING (IN-REPO SOURCE)

→ GIT SUBMODULES AND SUBTREES

Submodules are linked at pinned commits.
Both are useful in organizations sharing libraries across projects.

→ COPY SOURCE INTO REPOSITORY

More common for stable header only dependencies;
simplest but no automatic updates.

BUILD SYSTEMS

CMAKE

The de facto standard for C++ projects. Highly flexible and supported by all major IDEs. It is also a build system generator.

BAZEL, SCONS AND MESON

High-performance alternatives for large-scale projects and fast incremental builds.

MAKEFILES AND AUTOTOOLS

Autotools is useful in Unix-like systems, Linux, macOS and MinGW Windows.

STRATEGIES FOR USING C++ IN WEB BACKENDS

01. SPECIALIZED MICROSERVICE

> As optimized specialized services, communication through HTTP, gRPC or other IPC means.

02. FULL C++ BACKEND

> Primary application is C++; components can be C++ as well.

03. SPECIALIZED COMPONENTS IPC

> libraries called through an exposed C API from services written in other languages.

SCALING BEYOND THE SINGLE SERVER – DISTRIBUTED SYSTEMS

- Authentication, Authorization, Validation & Cryptography
- Event streaming & sourcing, message queues, PubSub, event driven architecture
- Servers, Serverless, Cloud, VPS, On Prem hosting.
- Database(SQL and No SQL) , DB federation, ACID, CAP, PACELC Indexes, Transactions, Sharding, Normalization & Denormalization, Replication.
- Distributed systems – Availability, Redundancy, service discovery , Scaling, Storage, circuit breaker, rate limiting, SLA, SLO, Disaster recovery, Monolith and Microservices.
- Observability & Monitoring.
- Caching, Clustering, Load balancing, CDN , Proxy.
- WebRTC, QUIC, WebSockets, Polling, and WebBidi

RATE LIMITING

```
1 #include <drogon/drogon.h>
2 #include <drogon/plugins/Hodor.h>
```

```
1 drogon::app().registerBeginningAdvice([]() {
2     auto user_id_getter =
3         [](const drogon::HttpRequestPtr &req) -> std::optional<std::string> {
4             std::optional<std::string> user_id_opt =
5                 req->getAttributes()->get<std::string>("current_user_id");
6
7             return user_id_opt->empty() ? std::nullopt : user_id_opt;
8         };
9     drogon::app().getPlugin<drogon::plugin::Hodor>()->setUserIdGetter(
10         user_id_getter);
11 });
12
```

```
1 {
2     "name": "drogon::plugin::Hodor",
3     "dependencies": [],
4     "config": {
5         // The algorithm used to limit the number of requests.
6         // The default value is "token_bucket". other values are "fixed_window"
7         // or "sliding_window".
8         "algorithm": "token_bucket",
9         // a regular expression (for matching the path of a request) list for
10        // URLs that have to be limited. if the list is empty, all URLs are limited.
11        "urls": [
12            "^/auth/.*",
13            "^/admin/.*",
14            "^/workflow/.*"
15        ],
16        // The time unit in seconds. the default value is 60.
17        "time_unit": 60,
18        // The maximum number of requests in a time unit. the default value 0
19        // means no limit.
20        "capacity": 0,
21        // The maximum number of requests in a time unit for a single IP. the
22        // default value 0 means no limit.
23        "ip_capacity": 5,
24        // The maximum number of requests in a time unit for a single user.
25        // a function must be provided to the plugin to get the user id from the request.
26        // the default value 0 means no limit.
27        "user_capacity": 5,
28        // The message body of the response when the request is rejected.
29        "rejection_message": "Too many requests",
30        // In seconds, the minimum expiration time of the limiters for different
31        // IPs or users. the default value is 600.
32        "limiter_expire_time": 60,
33        "sub_limits": [],
34        // Trusted proxy ip or cidr
35        "trust_ips": ["127.0.0.1"]
36    }
37 }
```

BEFORE WE START, SOME HELPERS...

```
1  /**
2   * @brief Strict read JSON that requires all keys to be present and does not
3   * allow unknown keys.
4   * @tparam T The type to read into.
5   * @param buffer The input buffer containing the JSON data.
6   * @return glz::expected containing the parsed value of type T or an error
7   * context
8   */
9  template <glz::read_supported<glz::JSON> T, glz::is_buffer Buffer>
10 [[nodiscard]] inline glz::expected<T, glz::error_ctx> strict_read_json(
11     Buffer &&buffer) {
12     T value{};
13     glz::context ctx{};
14     const glz::error_ctx ec = read<glz::opts{.error_on_missing_keys = true}>(
15         value, std::forward<Buffer>(buffer), ctx);
16     if (ec) {
17         return glz::unexpected<glz::error_ctx>(ec);
18     }
19     return value;
20 }
```

```
1  template <typename T>
2  [[nodiscard]] inline rfl::Result<T>
3  strict_read_msgpack(std::span<const char> buffer) {
4      return rfl::msgpack::read<T>(buffer);
5  }
```

```
1  template <typename T>
2  [[nodiscard]] inline std::expected<T, zpp::bits::error_code>
3  strict_read_binary(std::span<const std::byte> buffer) {
4      auto result = zpp::bits::from_bytes<T>(buffer);
5      if (result) {
6          return std::move(*result);
7      } else {
8          return std::unexpected(result.error());
9      }
10 }
```

EVENT DRIVEN ARCHITECTURE: TASK QUEUES

2

```
1 // Dispatch message to appropriate handler
2 drogon::app().getLoop()->queueInLoop(
3     [this, r_key = std::move(routing_key),
4     m_body = std::move(message_body),
5     delivery_tag = envelope.delivery_tag,
6     conn_ptr = rabbitmq_service.get_conn()]() mutable {
7     drogon::async_run([this, r_key = std::move(r_key),
8         m_body = std::move(m_body), delivery_tag,
9         conn_ptr]() mutable -> drogon::Task<void> {
10
11         bool should_ack = co_await dispatch_message(std::move(r_key),
12             std::move(m_body));
13
14         int ack_res{};
15         if (should_ack) {
16             ack_res = amqp_basic_ack(conn_ptr, AMQP_CHANNEL, delivery_tag,
17                 false);
18         } else {
19             ack_res = amqp_basic_nack(
20                 conn_ptr, AMQP_CHANNEL, delivery_tag, false,
21                 true); // requeue
22         }
23         if (ack_res < 0) {
24             LOG_ERROR << "An error occurred during ack/nack";
25         }
26         RabbitMQService::warn_on_amqp_error(
27             amqp_get_rpc_reply(conn_ptr), "Ack/Nack failed");
28         co_return;
29     });
30 });
```

1

```
1 void EventStream::start_consuming() {
2     queue_thread_ = std::jthread([this](std::stop_token token) {
3         try {
4             auto &rabbitmq_service =
5                 ServiceManager::get_instance().get_rabbitmq_service();
6
7             rabbitmq_service.declare_exchange(events_exchange, "topic");
8
9             rabbitmq_service.declare_queue(new_product_queue, true, false, false);
10            rabbitmq_service.bind_queue(new_product_queue, events_exchange,
11                new_product_queue);
12
13            rabbitmq_service.declare_queue(new_user_queue, true, false, false);
14            rabbitmq_service.bind_queue(new_user_queue, events_exchange,
15                new_user_queue);
16
17            consumer_tags_.emplace_back(rabbitmq_service.consume(new_product_queue));
18            consumer_tags_.emplace_back(rabbitmq_service.consume(new_user_queue));
19
20            while (!token.stop_requested()) { // Continuous consumption
21                amqp_maybe_release_buffers(rabbitmq_service.get_conn());
22                amqp_envelope_t envelope{};
23                amqp_rpc_reply_t rep = amqp_consume_message(rabbitmq_service.get_conn(),
24                    &envelope, nullptr, 0);
25
26                if (rep.reply_type != AMQP_RESPONSE_NORMAL) {
27                    if (rep.reply_type != AMQP_RESPONSE_NONE) {
28                        // Recovery strategy here
29                    }
30                    std::this_thread::sleep_for(std::chrono::milliseconds(1));
31                    continue;
32                }
33
34                std::string message_body(
35                    reinterpret_cast<char*>(envelope.message.body.bytes),
36                    envelope.message.body.len);
37                std::string routing_key(
38                    reinterpret_cast<char*>(envelope.routing_key.bytes),
39                    envelope.routing_key.len);
40
41                // Dispatch message to appropriate handler
42            }
43        } catch (const std::exception &e) {
44            // handle error
45        }
46    });
47 }
```

2

EVENT DRIVEN ARCHITECTURE: TASK QUEUES

3

```
1 drogon::Task<bool> EventStream::dispatch_message(std::string &&routing_key,  
2                                                  std::string &&message_body) {  
3     if (routing_key.starts_with(new_product_queue)) {  
4         auto msg = utilities::strict_read_json<NewProductMessage>(message_body);  
5         if (!msg) {  
6             LOG_ERROR << "Error parsing NewProductMessage: "  
7                 << msg.error().includer_error;  
8             co_return true; // don't requeue bad messages  
9         }  
10        co_return co_await handle_new_product(std::move(*msg));  
11    } else if (routing_key.starts_with(new_user_queue)) {  
12        auto msg = utilities::strict_read_json<NewUserMessage>(message_body);  
13        if (!msg) {  
14            LOG_ERROR << "Error parsing NewUserMessage: "  
15                << msg.error().includer_error;  
16            co_return true;  
17        }  
18        co_return co_await handle_new_user(std::move(*msg));  
19    } else {  
20        LOG_WARN << "Unknown routing key: " << routing_key  
21            << ", message: " << message_body;  
22    }  
23    co_return true;  
24 }
```

```
1 drogon::Task<bool> EventStream::handle_new_user(  
2     NewUserMessage &&message) {  
3     auto db = drogon::app().getDbClient();  
4     auto transaction = co_await db->newTransactionCoro();  
5     try {  
6  
7         // use transaction for atomic db queries  
8  
9         // publish to other services  
10  
11    } catch (const drogon::orm::DrogonDbException &e) {  
12        trans->rollback();  
13        // handle error  
14    } catch (const std::exception &e) {  
15        // handle error  
16    }  
17 }
```

```
1 void EventStream::stop_consuming() {  
2     if (checkout_thread_.joinable()) {  
3         checkout_thread_.request_stop();  
4         checkout_thread_.join();  
5     }  
6 }
```

RETRIES IN EVENT DRIVEN ARCHITECTURE

→ Dispatcher

```
1 } else if (routing_key.starts_with(retry_prefix)) {
2     auto msg =
3     utilities::strict_read_json<RetryableMessage>(message_body);
4     if (!msg) {
5         LOG_ERROR << "Error parsing RetryableMessage: "
6                 << msg.error().includer_error;
7         co_return true;
8     }
9     co_await
10    publish_message(msg->original_exchange, msg->original_routing_key,
11                   msg->original_payload);
12    co_return true;
13 }
```

→ Consuming thread

```
1 // Delayed exchange for timeouts and retries
2 std::array<amqp_table_entry_t, 1> delayed_exchange_args = {
3     amqp_table_entry_t{
4         .key = amqp_cstring_bytes("x-delayed-type"),
5         .value = {.kind = AMQP_FIELD_KIND_BYTES,
6                 .value = {.bytes = amqp_cstring_bytes("topic")}}}};
7
8 rabbitmq_service.declare_exchange(delayed_exchange, "x-delayed-message",
9                                  true, false,
10                                  std::span(delayed_exchange_args));
11
12 rabbitmq_service.declare_exchange(dead_retries_exchange, "topic");
13
14 std::array<amqp_table_entry_t, 2> dead_queue_dlx_args = {
15     amqp_table_entry_t{
16         .key = amqp_cstring_bytes("x-dead-letter-exchange"),
17         .value = {.kind = AMQP_FIELD_KIND_UTF8,
18                 .value = {.bytes = amqp_cstring_bytes(
19                     dead_retries_exchange.data())}}},
20     amqp_table_entry_t{
21         .key = amqp_cstring_bytes("x-dead-letter-routing-key"),
22         .value = {.kind = AMQP_FIELD_KIND_UTF8,
23                 .value = {.bytes = amqp_cstring_bytes(
24                     dead_retries_queue.data())}}}};
25
26 rabbitmq_service.declare_queue(retry_queue, true, false, false,
27                               dead_queue_dlx_args);
28 rabbitmq_service.bind_queue(retry_queue, delayed_exchange,
29                             make_routing_key(retry_prefix, "#"));
30
31 rabbitmq_service.consume(retry_queue);
```

RETRIES IN EVENT DRIVEN ARCHITECTURE

→ schedule retry?

→ why no coroutine in catch block?

```
1  drogon::Task<bool> EventStream::handle_new_order(NewUserMessage &&message) {
2      auto db = drogon::app().getDbClient();
3      auto transaction = co_await db->newTransactionCoro();
4      try {
5
6          bool status_ok = co_await update_order_status(
7              trans, message.order_id, "processing_order", message.user_id);
8          if (!status_ok) {
9              trans->rollback();
10             co_await schedule_retry(events_exchange, new_order_queue, message,
11                                     "failed to update basket status",
12                                     update_status_max_retries);
13
14             co_return true;
15         }
16
17         ProcessPaymentMessage payment_msg{.retry_count = 0,
18                                           .max_retries =
19                                             payment_process_max_retries,
20                                           .basket_id = message.order_id};
21
22         co_await publish_message(payments::payments_exchange,
23                                 process_order_payment_queue,
24                                 glz::write_json(payment_msg).value_or(""));
25     } catch (const drogon::orm::DrogonDbException &e) {
26         trans->rollback();
27         schedule_retry_async(events_exchange, new_order_queue, message,
28                             e.base().what(), db_exception_max_retries);
29
30         co_return true;
31     } catch (const std::exception &e) {
32         if (trans)
33             trans->rollback();
34         std::string err = e.what();
35         schedule_retry_async(events_exchange, new_order_queue, message,
36                             e.base().what(), std_exception_max_retries);
37
38         co_return true;
39     }
40 }
```

→ handler with retry logic

RETRIES IN EVENT DRIVEN ARCHITECTURE

→ Concept for RetryMessage structs

```
1 template <typename T>
2 concept HasRetryInfo = requires(T t) {
3     { t.retry_count } -> std::same_as<uint32_t>;
4     { t.max_retries } -> std::same_as<uint32_t>;
5 };
```

```
1 /**
2  * Schedule message for reprocessing.
3  * Callers can pass a new max retry or we use already contained max retries in
4  * payload.
5  */
6 template <HasRetryInfo T>
7 inline static drogon::Task<void>
8 schedule_retry(std::string_view original_exchange,
9               std::string_view original_routing_key, T&& payload,
10               std::string_view error_log,
11               uint32_t new_max_retries = std::numeric_limits<uint32_t>::max(),
12               BackoffStrategy strategy = BackoffStrategy::MEDIUM) {
13     new_max_retries = std::min(payload.max_retries, new_max_retries);
14
15     if (payload.retry_count >= new_max_retries) {
16
17         DeadLetterPayload dead_payload{
18             .original_exchange = original_exchange,
19             .original_routing_key = original_routing_key,
20             .original_payload = glz::write_json(payload).value_or(""),
21             .error_log = error_log};
22
23         std::string dead_payload_string =
24             glz::write_json(dead_payload).value_or("");
25
26         co_await publish_message(dead_retries_exchange, dead_retries_queue,
27                                 dead_payload_string);
28     }
29
30     ++payload.retry_count;
31
32     RetryableMessage retry_msg{
33         .original_exchange = original_exchange,
34         .original_routing_key = original_routing_key,
35         .original_payload = glz::write_json(payload).value_or(""),
36         .retry_count = payload.retry_count,
37         .max_retries = new_max_retries};
38
39     std::string retry_payload = glz::write_json(retry_msg).value_or("");
40
41     uint64_t delay_ms = calculate_backoff(payload.retry_count, strategy);
42
43     co_await EventStream::publish_message(
44         delayed_exchange, retry_prefix, retry_payload, delay_ms);
45     co_return;
46 }
```

CASE STUDY: SEMANTIC SEARCH ENGINE

→ Proof of concept: in a Colab or Jupyter Notebook

PROOF OF CONCEPT (PYTHON)

PYTORCH
SAFE TENSORS MODEL

FAISS

```
1 from sentence_transformers import SentenceTransformer
2
3 # Initialize the model
4 model = SentenceTransformer("vectopath/SearchMap_Preview",
5                             trust_remote_code=True)
6
7 embeddings = model.encode(all_product_data, show_progress_bar=True)
```

```
1 import numpy as np
2 import faiss
3
4 # Create FAISS index
5 embedding_dim = 1024
6 index = faiss.IndexFlatL2(embedding_dim)
7
8 # Add product embeddings
9 product_embeddings = model.encode(product_descriptions, show_progress_bar=True)
10 index.add(np.array(product_embeddings).astype('float32'))
11
12 # Search
13 query_embedding = model.encode([query])
14 distances, indices = index.search(
15     np.array(query_embedding).astype('float32'),
16     k=10
17 )
18
```

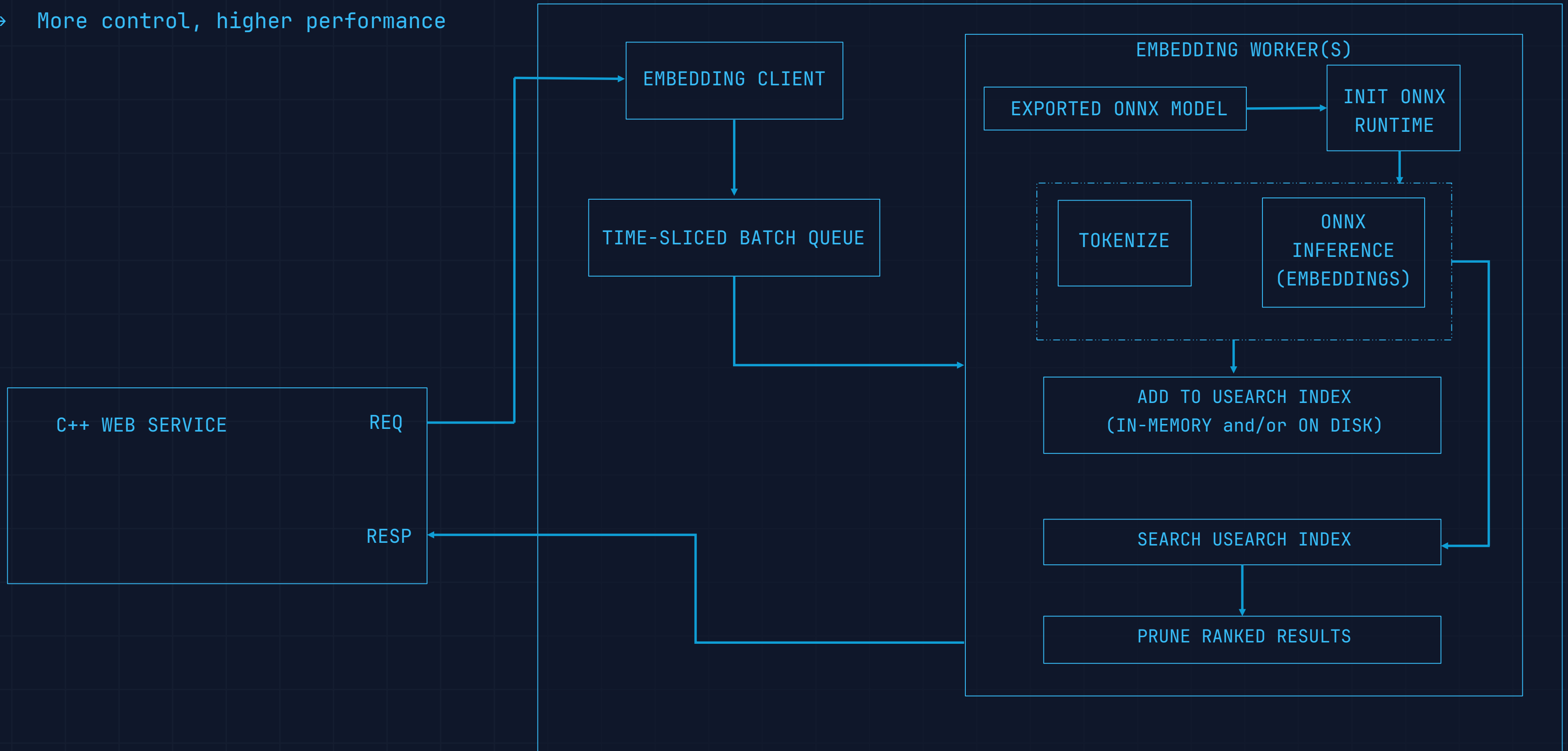
CASE STUDY: SEMANTIC SEARCH ENGINE

→ Python Webserver + C++ Engine



CASE STUDY: SEMANTIC SEARCH ENGINE

→ More control, higher performance

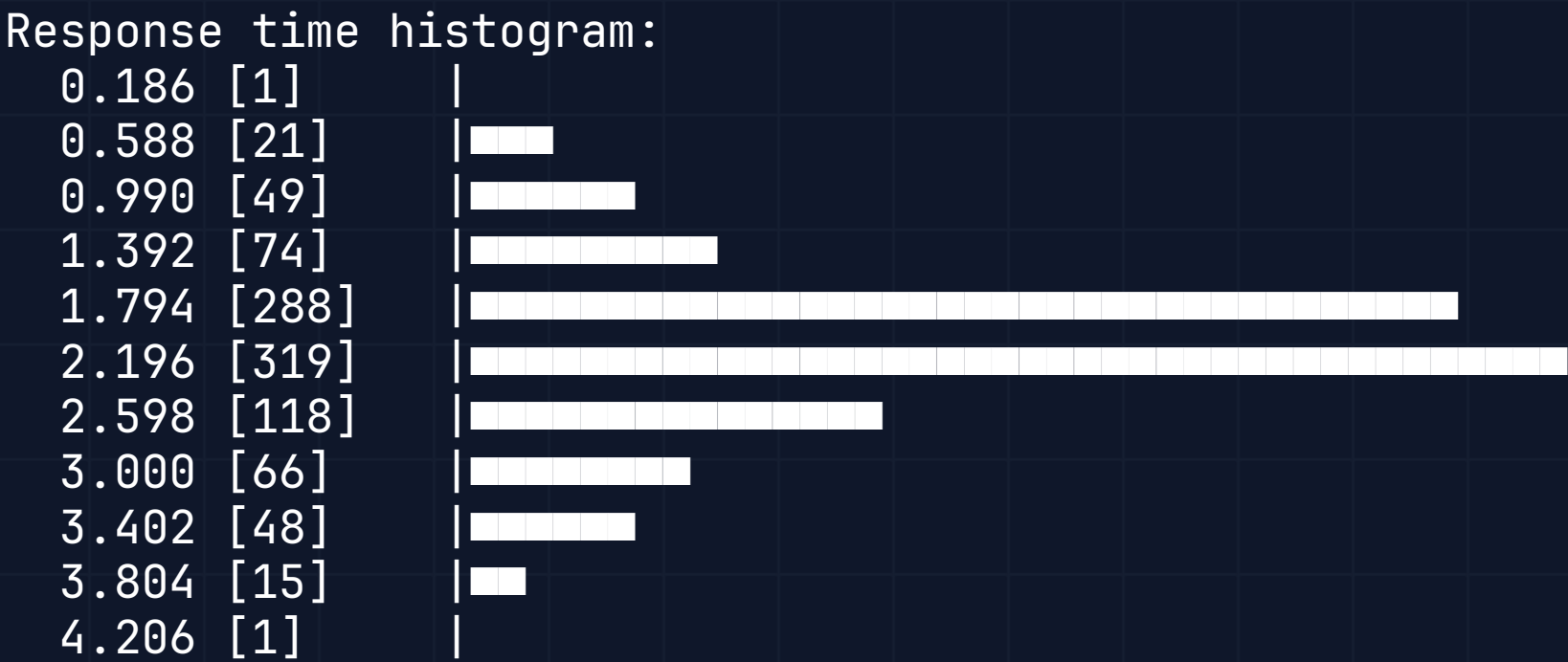


SEMANTIC SEARCH ENGINE RESULTS

```
./hey.exe -n 1000 -m POST -H 'Content-Type: application/json' -d '{ "query": "iphone 15 pro", "geo": { "lat": 33.748, "lon": -112.0, "radius_km": 20 }, "limit": 20, "offset": 0}' http://localhost:5555/products/search
```

→ Without time batching

Summary:
Total: 38.6461 secs
Slowest: 4.2059 secs
Fastest: 0.1856 secs
Average: 1.8992 secs
Requests/sec: 25.8758
Total data: 666012 bytes
Size/request: 666 bytes



→ With time batching

Summary:
Total: 9.1053 secs
Slowest: 0.8316 secs
Fastest: 0.0836 secs
Average: 0.4506 secs
Requests/sec: 109.8262
Total data: 666746 bytes
Size/request: 666 bytes



CASE STUDY: SEMANTIC SEARCH ENGINE RESULT

```
./hey.exe -n 1000 -m POST -H 'Content-Type: application/json' -d '{ "query": "iphone 15 pro", "geo": { "lat": 33.748, "lon": -112.0, "radius_km": 20 }, "limit": 20, "offset": 0}' http://localhost:5555/products/search
```

Latency distribution:

10% in 1.1888 secs
25% in 1.6094 secs
50% in 1.8327 secs
75% in 2.1879 secs
90% in 2.7643 secs
95% in 3.0361 secs
99% in 3.4681 secs

Details (average, fastest, slowest):

DNS+diAlup: 0.0157 secs, 0.1856 secs, 4.2059 secs
DNS-lookup: 0.0003 secs, 0.0000 secs, 0.0066 secs
req write: 0.0001 secs, 0.0000 secs, 0.0080 secs
resp wait: 1.8829 secs, 0.1115 secs, 3.8878 secs
resp read: 0.0003 secs, 0.0000 secs, 0.0055 secs

Latency distribution:

10% in 0.4130 secs
25% in 0.4267 secs
50% in 0.4346 secs
75% in 0.4439 secs
90% in 0.4579 secs
95% in 0.4740 secs
99% in 0.8237 secs

Details (average, fastest, slowest):

DNS+diAlup: 0.0164 secs, 0.0836 secs, 0.8316 secs
DNS-lookup: 0.0005 secs, 0.0000 secs, 0.0110 secs
req write: 0.0002 secs, 0.0000 secs, 0.0079 secs
resp wait: 0.4338 secs, 0.0834 secs, 0.4923 secs
resp read: 0.0001 secs, 0.0000 secs, 0.0021 secs

SUMMARY

- Building powerful web services and backends easily in C++ is accessible today.
- There are benefits to building them.
- Great performance at a lower cost in your deployments.

THANK YOU

Connect with me:



<https://github.com/uzoochogu>

<https://discord.gg/sdQHmYfq>

C++ for High Performance Web Applications



Why C++ for Web Applications ?

- Predictable and deterministic Performance
- Speed – e.g. fastest AWS lambda cold start!
- Resource Efficiency
- Low Latency
- Access to Powerful Libraries and Direct code reuse
- Hardware Utilization

Notable C++ Web Frameworks and Libraries?

framework	summary	features	speed	prod ready
Drogon	High performance framework. Full featured.	★★★★	🚀🚀🚀🚀	✅
Lithium	High performance, easy to use. Linux only.	★★★	🚀🚀🚀🚀	✅
Oatpp	Modern, easy to use, well documented.	★★★★	🚀🚀🚀🚀	✅
CrowCpp	Inspired by Python Flask. elegant.	★★★	🚀🚀🚀	✅
Userver	Stackful coroutine based, extremely feature rich.	★★★★★	🚀🚀🚀🚀🚀	✅
Mongoose	Light weight. Good for embedded systems	★★★★	🚀🚀🚀	✅
ffead	High performance, extensive features. Pluggable backends	★★★★★	🚀🚀🚀🚀🚀	✅
Glaze (networking)	Blazing fast serialization. many networking features	★★★	?	✅
vixcpp	Modular arch, good ergonomics.	★★★★	🚀🚀🚀	✅
metaspex	Enterprise-ready backend development platform	★★★★★	?	✅

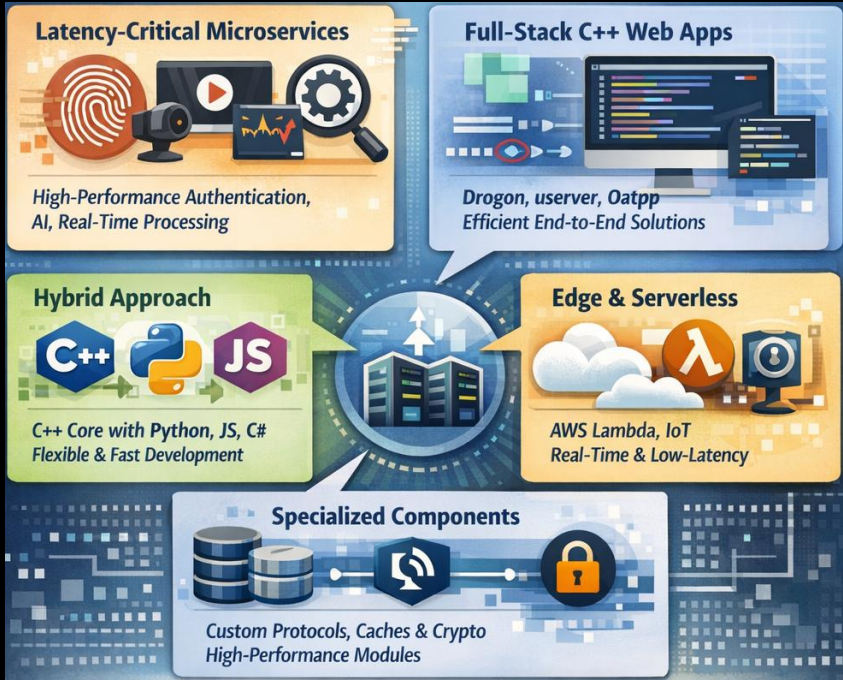
→ See [TechEmpower benchmarks](#) for performance comparisons.

Package Management

- vcpkg
- conan
- CMake's FetchContent - see this guide
- Git submodules
- Direct copy in repo - Good for extremely stable headers and utilities. Popularly used with single-header libraries.

Tip: Lock dependencies to specific versions or commits to avoid supply chain risks. Test thoroughly when upgrading.

Strategies for using C++ in Web Backends



Choosing Libraries: What to consider

- Community & Adoption: Popular, well-maintained libraries are safer bets.
- Performance: Benchmarks and real-world usage. Always benchmark for your use case.
- Security: Active patching and secure defaults.
- Features
- Ease of Use: Good documentation and examples.
- C++ Standard Support: Prefer libraries using modern C++ idioms(can help with safety, low level APIs may be more performant though, mix and benchmark).
- License: Ensure compatibility with your project

Further reading and Research

- CppCon, CppOnline, CppNow, MeetingC++
- [Learn CMake](#)
- [awesome-cpp](#)
- Reddit r/cpp

Summary

Modern C++ empowers you to build web applications and services that are fast, efficient, and robust—provided you follow best practices and leverage the right tools. C++ is not just for systems programming—it is a first-class choice for high-performance web backends. Whether you use it for critical microservices, full-stack APIs, or specialized components, C++ delivers unmatched speed, efficiency, and control for demanding web workloads.

Common libraries for web services

- JSON: [nlohmann/json](#), [RapidJson](#), [Glaze](#), [picojson](#), [yyjson](#), [simdjson](#), [Jsonifier](#), [JsonCpp](#), [Boost.Json](#)
- Cryptography: [argon2](#), [libsodium](#), [jwt-cpp](#)
- Networking: [Boost.Asio](#), [ZeroMQ](#)
- Database Drivers and ORMs: [libpqxx](#) (Postgres), [mongo-cxx-driver](#) or the driver embedded in your web framework. [TinyORM](#).
- Validation: Validation can be handled through custom parsers, so you can use parser combinator libraries like [Boost.parser](#) or [Lexy](#).
- Utilities and Standard Library: [Boost](#), [unordered_dense](#) hashmap, [Folly](#), [Poco](#), [Abseil](#), [EASTL](#)
- Event streaming, Message Queues and In-memory storage: [redis-plus-plus](#), [hiredis](#). These Redis compatible APIs allow you to use most popular in-memory stores like Pogocache, DragonFly and Valkey. Other libraries are [redpanda](#), [rabbitmq-c](#), [AMQP-CPP](#), [cppkafka](#), [librdkafka](#), [modern-cpp-kafka](#)
- ML, Math, Matrices - [Eigen](#), [mlpack](#), [armadillo](#), GLM
- Identifiers - [boost-uuid](#)
- Vector search – [usearch](#), [zvec](#)
- ML and GPU operations – cudnn, cuda, cublas, [parrot](#), vulkan-compute, [Kompute](#), pytorch, ONNX, TensorRT.
- Cloud SDKs: [Google Cloud C++](#), [AWS C++ SDK](#), [AWS Lambda C++ Runtime](#), [Azure C++ SDK](#).
- Serialization – [zpp::bits](#), glaze, Cap'n Proto, flatbuffers,

Web Backend concepts

- Authentication, Authorization, Validation & Cryptography
- Event streaming & sourcing, message queues, PubSub, event driven architecture
- Servers, Serverless, Cloud, VPS, On Prem hosting.
- Database(SQL and No SQL) , DB federation, ACID, CAP, PACELC Indexes, Transactions, Sharding, Normalization & Denormalization, Replication
- Distributed systems – Availability, Redundancy, service discovery , Scaling, Storage, circuit breaker, rate limiting, SLA, SLO, Disaster recovery, Monolith and Microservices.
- Observability & Monitoring, [DevOps](#)
- Caching, Clustering, Load balancing, CDN , Proxy
- OSI model and layers
- REST, GraphQL, gRPC
- Long polling, WebSockets, SSE, WebBidi

Good practices

- Use modern C++ memory safe and less error prone features (smart pointers, std::optional, std::expected, containers).
- Reserve exceptions for truly exceptional cases.
- Prefer prepared statements or constant strings for database queries to avoid SQL injection.
- Write comprehensive tests and use sanitizers (ASan, UBSan).
- Monitor for deadlocks, especially when using coroutines or async code.
- Study the documentation and source code of dependencies. The source code is the single source of truth. Documentation can be outdated. AI can be outdated or just wrong.
- When confused, benchmark, monitor for performance regressions, profile hot paths, benchmark different approaches: Google Benchmark is a nice micro-benchmarking lib. Compiler explorer is a good tool to quickly test patterns and compare compiler output. Other useful tools are Quickbench), Cpp Insights (See your source code with the eyes of a compiler).
- Spend time to setup your dev environment, LSPs, debuggers are really useful. Lots of time is spent debugging.